Course Script

INF 5110: Compiler construction

INF5110, spring 2021

Martin Steffen

Contents

Chapter 6 Symbol tables

What is it about?

Learning Targets of this Chapter

1. symbol table data structure
2. design and implementation choices
3. how to deal with scopes
4. connection to attribute grammars

Contents

6.1 Introduction

6.1.1 Symbol tables, in general

- **central** data structure
- "data base" or repository associating properties with "names" (identifiers, symbols)[1]
- **declarations**
  - constants
  - type declarationss
  - variable declarations
  - procedure declarations
  - class declarations
  - . . .
- *declaring* occurrences vs. *use* occurrences of names (e.g. variables)

- goal: associate attributes (properties) to syntactic elements (names/symbols)
- storing once calculated: (costs memory) $\leftrightarrow$ recalculating on demand (costs time)
- most often: **storing** preferred
- but: can't I store it in the nodes of the *AST*?
  - remember: *attribute grammar*
  - however, fancy attribute grammars with many rules and complex synthesized/inherited attribute (whose evaluation traverses up and down and across the tree):

[1] Remember the (general) notion of "attribute".

    * might be intransparent
    * storing info *in* the tree: might not be efficient

⇒ central repository (= **symbol table**) better

6.1.2 So: do I need a symbol table?

In theory, alternatives exists; in practice, yes, symbol tables is the way to go; most compilers do use symbol tables.

Most often (and in our course), the symbol table is set up once, containing all the symbols that occur in a given program, and then, the semantic analyses (type checking, etc.) update the table accordingly. The symbol table are "static" in that they are part of the compiler, but not the run-time system. There are also some languages, which allow "manipulation" of symbol tables at *run time* (Racket is one (formerly PLT scheme)).

6.2 Symbol table design and interface

6.2.1 Symbol table as abstract data type

- separate **interface** from implementation
- ST: "nothing else" than a lookup-table or *dictionary*
- associating "keys" with "values"
- here: keys = names (id's, symbols), values the attribute(s)

Schematic interface: two core functions (+ more)

- *insert*: add new binding
- *lookup*: retrieve

besides the core functionality:

- structure of (different?) *name spaces* in the implemented language, *scoping* rules
- typically: not one single "flat" namespace ⇒ typically not one big *flat* look-up table

⇒ influence on the design/interface of the ST (and indirectly the choice of implementation)

- necessary to "delete" or "hide" information (*delete*)

A symbol table is, typically, not just a "flat" dictionary, neither conceptually nor the way it's implemented. *Scoping* typically is something that often complicates the design of the symbol table.

It should also be clear from the context of the discussion: when we speak of the *value* of an attribute we typically don't mean the ultimate run-time value of the symbol, like the concrete integer run-time value of an expression. The value of an attribute is meant in the "meta"-way, the value that the analysis attaches to the entity, for instance its type, its address, etc. (and only in rather rare cases, its run-time level value). The situation is the same as for attribute grammars and indeed, symbol tables can be seen as a data structure realizing “attributes”. See also the next slide, contrasting two ways of attaching “attributes” to entities in a (syntax) tree: “internal”, as part of the nodes, or external, in a separate repository (known as symbol table).

The attached information, though, is mostly *semantic* in nature. After all, in the current phase, we have left the syntact analysis phase behind us, the parser, and doing *semantic analysis*. For instance, when attaching a type like integer to a symbol, it *is* a semantic value in the sense that it carries semantic infomation. It’s a static abstraction of the possible concrete semantic values that can be stored at run-time in the corresponding variable (and where the value may of course change in an imperative language).

6.2.2 Two main philosophies

traditional table(s)

- central repository, separate from AST
- interface
  - $lookup(name)$,
  - $insert(name, decl)$,
  - $delete(name)$
- last 2: update ST for declarations *and* when entering/exiting *blocks*

decls. in the AST nodes

- do look-up $\Rightarrow$ tree-*search*
- insert/delete: implicit, depending on relative positioning in the tree
- look-up:
  - efficiency?
  - however: optimizations exist, e.g. “redundant” extra table (similar to the traditional ST)

Here, for concreteness, *declarations* are the attributes stored in the ST. In general, it is not the only possible stored attribute. Also, there may be more than one ST.

Language often have different “name spaces”. Even a relatively old-school language like C has 4 different name spaces for identifiers. There are different kinds of identifiers, and different rules (for instance wrt. scoping) apply to them. One way to arrange them could be to have different symbol tables, one specially for each name space. Later we will also have the situation (but not caused by different kinds of identifiers), where the symbol table is arranged in such a way that smaller symbol tables (per scope) are linked together where a symbol table of a “surrounding” scope points to a symbol table representing a scope nested deeper. One might see that as having “many” symbol tables, but maybe that’s misleading. It’s more an internal representation with a linked structure, but that data structure containing many individiual table is better seen conceptually as one symbol table (*the* symbol table of the language), but one with a complex behavior reflecting the lexical scoping of the language. Actually, whether or not one implements it in chaining up a bunch of individual hash tables or similar structures or doing a different representation, is a design choice, both realizing the same external behavior at the interface. In that spirit, also the remark that C has 4 different name spaces (which is true) and therefore a C compiler may make use of 4 symbol tables is a matter of how one sees (and implements) it: one may as well see and implement it as one symbol table (with 4 different kinds of identifiers which are treated differently).

A cautionary note: You may find the statement that C (being old-fashioned) does not feature name spaces. The discussion here was about the internal organization and scoping rules for various classes of identifers in C, which form internally 4 different name spaces. But C does not have elaborate *user-level* mechanisms to introduce name spaces; therefore, one may stumble upon statements like “C does not support name spaces”...

6.3 Implementing symbol tables

6.3.1 Data structures to implement a symbol table

- different ways to implement *dictionaries* (or look-up tables etc.)
  - simple (association) lists
  - trees
    - balanced trees (AVL, B, red-black, binary-search trees)
  - **hash** tables, often method of choice
  - functional vs. imperative implementation
- careful choice influences efficiency
- influenced also by the language being implemented
- in particular, by its **scoping** rules (or the structure of the name space in general) etc.[2]

6.3.2 Nested block / lexical scope

for instance: *C*

```
{ int i; ... ; double d;
  void p(...);
  {
     int i;
     ...
  }
  int j;
  ...
```

more later

[2]Also the language used for implementation (and the availability of libraries therein) may play a role.

6.3.3 Blocks in other languages

TEX

```
\def\x{a}
{
  \def\x{b}
  \x
}
\x
\bye
```

LATEX

```
\documentclass{article}
\newcommand{\x}{a}
\begin{document}
\x
{\renewcommand{\x}{b}
  \x
}
\x
\end{document}
```

But remember: static vs. dynamic binding (see later)

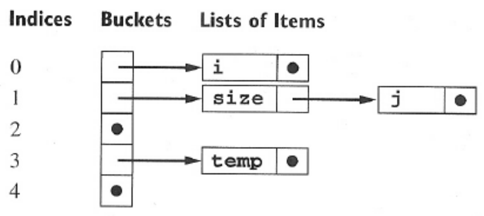
LATEX and TEX are chosen for easy trying out the result oneself (assuming that most people have access to LATEX and by implication, TEX). TEX is the underlying "core" on which LATEX is put on top. There are other formats on top of TEX (`texi` is another one; `texi` is involved, for instance, type setting the pdf version of the Compila language specification).

6.3.4 Hash tables

- classical and common implementation for STs
- "hash table":
  - generic term itself, different general forms of HTs exists
  - e.g. *separate chaining* vs. *open addressing*

There exists alternative terminology (cf. INF2220 in the older numbering scheme, it's the algo & data structures lecture), under which separate chaining is also known as *open hashing*. The *open addressing* methods are also called *closed hashing*. It's confusing, but that's how it is, and it's just words.

Separate chaining



```
{
  int temp;
  int j;
  real i;
  void size (....) {
    {
       ....
    }
  }
}
```

The example is a rather trivial one. It illustrates one basic fact about hash-table as underlying data structure for dictionaries. Namely the fact that there can be *hash conflicts*. In the example, it's assumed that there is such a hash-conflict between the identifiers `size` and `j`. A hash conflict means, when applying the chosen hash-function onto 2 keys (here the two identifiers or symbols), the resulting hash values is identical. That's the conflict. One design of hash tables deals with the situation by arranging entries of conflicting hashes into a linked list. That's what is illustrated here and that techniques is called *separate chaining*.

Conventionally, the compiler would treat the program from beginning to the end, wich means it would enter the information about `j` before doing the same for `size`. It is, in that situation of a linked list, plausible that, after adding both pieces of information in a linked list arrangement, the `size`, being entered last, appears at the head of the list.

We also see that the identifiers for integer variables `i`, `j`, etc. are treated the same way as those for procedures (here `size`). In particular they are added to the same symbol table. There is nothing wrong with that, but, as mentioned before, sometimes identifiers are grouped in different name spaces or, in general, the use of names is generally more complex, so one may choose to handle things with different symbol tables (for different purposes of classes of identifiers etc.)

An important complicating factor for managing names or identifies is the concept of *scope*. Identifiers typcially occur in scopes; in the discussion here, I focus on static or lexical scopes. It's an "area" or "portion" of the of of a programe, where an identifier "lives". Note that it's not about the lifetime of the value stored in the variable. Quite later in the lecture, we will talk about liveness of variables, and *there*, being "live" at some point in the program means the current *value* in the variable will or maybe be used in the future. Figuring out information about that is called *liveness analysis*, but it's for a later chapter. Here it's about (lexical) scopes which designates the area in the code where the variable itself "exists" and where it even makes sense to ask about the value of a variable variable or whether it's live outside the variable's scope.

Of course the notions of scopes and liveness are somehow connected connection. If the a program execution proceeds in that it "leaves" the scope of a variable, the variable certainly becomes is dead in that sense: not only the variable's value will not be used in the future, the whole variable has moved "out of scope", so to say.

6.3.5 Block structures in programming languages

- almost no language has one global namespace (at least not for variables)
- pretty old concept, seriously started with ALGOL60

Block

- "region" in the program code
- delimited often by `{` and `}` or `BEGIN` and `END` or similar
- organizes the **scope** of declarations (i.e., the name space)
- can be **nested**

6.3.6 Block-structured scopes (in C)

```
int i, j;

int f(int size)
{ char i, temp;
  ...
  { double j;
    ..
  }
  ...
  { char * j;
    ...
  }
}
```

6.3.7 Nested procedures in Pascal

```
program Ex;
var i,j : integer

function f(size : integer) : integer;
var i, temp : char;
   procedure g;
   var j : real;
   begin
      ...
   end;
   procedure h;
```

```
    var j : ^char;
    begin
       ...
    end;

begin (* f's body *)
  ...
end;
begin  (* main program *)
    ...
end.
```

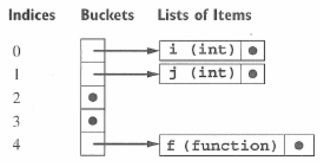
The Pascal-example shows a feature of Pascal, which is *not* supported by C, namely nested declarations of functions or procedures. As far as scoping and the discussion at the current point in the lecture is concerned, that's not a big issue: just that concerning names for variables, C and Pascal allow nested blocks, but for names representing functions or procedures, Pascal offers more freedom.

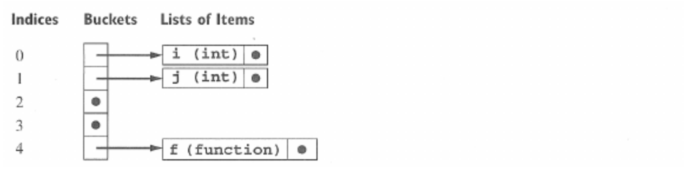
The scoping rules of a programming language influences the design and implementation of a language's symbol table, that's a general message here. Later we will see how scoping also influences the design of the so-called *run-time environments*. Also in that part, we will discuss how the run-time environments for Pascal are more complex than those for C. And scoping regimes more complex than that of Pascal add even more complexity for symbol tables and in particular to the runtime environment.

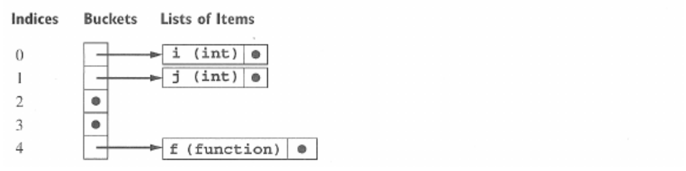
6.3.8 Block-strucured via stack-organized separate chaining

```
int i, j;

int f(int size)
{ char i, temp;
  ...
  { double j;
     ..
  }
  ...
  { char * j;
     ...
  }
}
```





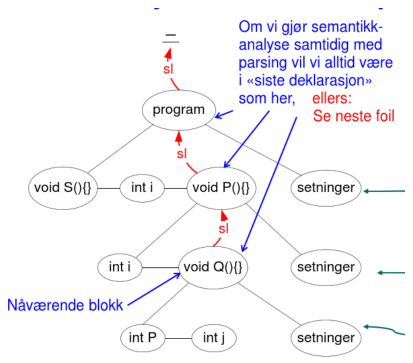


The 3 pictures correpond to three “points” inside the C program from above. The first one after entering the scope of function `f`. When saying "entering", it’s not (only) meant at run-time when calling the function, it’s meant when the analysis starts processing the body of the procedure. Inside the body of the function (immediately after entering), the two local variables are available, and of course also the formal parameter `temp`, which can be seen as a local variable, as well. At that point, the global variable `i` of type `int` is no longer “visible” or accessible, any reference to `i` will refer to the local variable `i` at that point.

Upon entering the first nested local scope, a second variable `j` is entered (making the global variable `j` inaccessible). That situation is *not* shown in the pictures. New, when *leaving* the mentioned scope, one way of dealing with the situation is that the additional second `j` of type `double` is *removed* from the hash-table again (shortening the corresponding linked chain). What is shown is a situation inside the *second* nested scope with another variable `j` (now a char pointer). Since the first nested local scope has been left at that point, the corresponding `j` “has become history”, and the hash table of the third picture only contains the global `j` variable (which is unaccessible) and the now relevant second local `j` variable.

6.3.9 Using the syntax tree for lookup following (static links)

```
lookup (string n) {
   k = current , surrounding block
   do              // search for n in  decl for block k;
      k = k.sl     // one nesting level  up
   until found or k == none
}
```



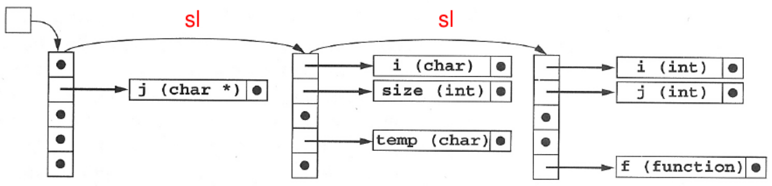
The notion of *static link* will be discussed later, in connection with the so-called run-time system and the run-time *stack*. There we go into more details, but the idea is the same as here: find a way to "locate" the relevant scope. If they are nested, connect them via some "parent pointer", and that pointer is known as static links (again, different names exists for that, unfortunately).

6.3.10 Alternative representation

- arrangement different from 1 table with stack-organized external chaining
- each *block* with its **own** hash table.
- standard hashing within each block
- **static links** to link the block levels

⇒ "tree-of-hashtables"

- AKA: *sheaf-of-tables* or *chained symbol tables* representation



Note that the top-most scope is at the right-hand side of the table, and the static-link always points to the (uniquely determined) surrounding scope (if any).

One may more generally say for this represention: it's one *symbol table* per block, as this form of organization can generally be done for symbol tables data structures where hash tables is just one of many possible data structure to implement look-up tables.

6.4 Block-structure, scoping, binding, name-space organization

6.4.1 Block-structured scoping with chained symbol tables

- remember the *interface*

- look-up: following the static link (as seen)
- **Enter** a block
  - create new (empty) symbol table
  - set static link from there to the "old" (= previously current) one
  - set the current block to the newly created one
- at **exit**
  - move the *current block* one level up
  - note: no *deletion* of bindings, just made *inaccessible*

As mentioned in the previous section: The notion of static links will be encountered later again when dealing with *run-time* environments and for analogous purposes: identfying lexical scopes in "block-stuctured" languages.

6.4.2 Lexical scoping & beyond

- block-structured lexical scoping: **central** in programming languages (ever since ALGOL60 ...)
- but: other scoping mechanisms exist (and exist side-by-side)
- example: C^{++}
  - member functions *declared* inside a class
  - *defined* outside
- still: method supposed to be able to access names defined in the *scope of the class* definition (i.e., other members, e.g. using `this`)

C^{++} class and member function

```
class A {
  ... int f(); ... // member function
}

A::f() {}    // def. of f ``in'' A
```

Java analogon

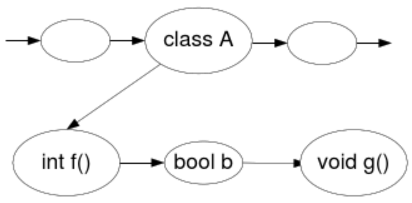
```
class A {
    int f() {...};
    boolean b;
    void h() {...};
}
```

6.4.3 Scope resolution in C^{++}

- class *name* introduces a **name for the scope**[3] (not only in C^{++})
- scope resolution operator **::**
- allows to explicitly refer to a "scope"'

[3]Besides that, class names themselves are subject to scoping themselves, of course ...

- to implement
  - such flexibility,
  - also for *remote access* like `a.f()`
- declarations are kept separately for each block (e.g. one hash table per class, record, etc., appropriately chained up)



6.4.4 Same-level declarations

Declarations of entities (variables, procedures, classes . . .) occur inside one scope. Global declarations can be consider to occur in one surrounding global scope. One scope contains normally multiple declarations. Since scopes can be nested, any declarations in the inner scope also belong to the other scope. Characteristic for declarations in nested blocks is that, inside the inner scope, the declarations take precedence over those in the surrounding block, should there be declarations using the same name. It's always the closest declaration (in terms on the nesting structure of scopes) that counts, as explained.

In the following we discuss how to treat *same-level* declarations inside one scope, focusing mainly on two alternatives: sequential or simultaneous (or collateral).

Multiple declarations of the same name

Before we look at those two alternatives, let's discuss other aspects concerning same-level declarations. One is how to treat multiple declarations of the *same* identifier at the same level. Like in the situation in Listing 6.1, where `i` is used for an integer variable as well as name of a type.

```
typedef int i
int i;
```

Listing 6.1: Same level declarations

One simple way is to simply *forbid* it. For instance in C, it (largely) forbidden; it's a bit more complex there, distinguising between declarations, declarations combined with definitions, tentative declarations . . . , but it's not relevant for us. Forbidding multiple declarations or definitions is also the solution followed in the *compila21* language of the oblig. That can of course be easily achieved: Before using the *insert* procedure to add a binding, one simply checks fist (with, say *lookup*) whether such a binding for the name exists already.

Another option is to allow it. The declarations show up in the block in some textual order, and then the convention is that a declaration make a previous one with the same name inaccessible. It's as if a new nested scope is introduced implicitly which start at the new declaration and lasts till the end of the "official" surrounding scope.

Those treatments can be more involved or fine-grained. For instance how to handle multiple declarations using the same identiier, but concerning different kinds of language elements. Like declaring a variable and a procedure with the same name, declaring a variables and introducing a type with the same name etc. Again, one could forbid such practice, or at least partially, for simplicity. It may even be that the language would not even allow to write multiple declations of that kind, because of lexical conventions of the language. For insance, names of types have to start with a capital letter and variables have to start with a lower case letter. So already the lexer (together with the parser) makes it impossible to declare a variable and a type with the same name.

Of course there is only so much a lexer (and a parser) can do. Typically, one has to deal with more than two categories of names (like variable names on the one hand and type names on the other) that one could allow to live side by side at the same level and it would be ridiculous to come up with a lexical scheme like the following: variable names and function names all start with lower case letters, but variables start with a letter from $a \dots l$ and function names start in the range $m \dots z$. Doable, and making multiple declarations from difference impossible, but nonsense, of course.

As a side remark: When one wishes, for instance to distinguish variables names from function names, one has to take into account what the programming language actually supports why programming with variables and functions. In many languages, variables are variables and functions are functions. However, there is the concept of function variables. In particular, in functional languages, there is no differences between functions and (other values). Consequently there is no difference conceptually between names for "conventional" values like integers, and function abstractions, which counts among values as well. So corresponding names would live in the same name space.

Overloading

If one want to allow that, say, functions and variables carry the same name when declared at the same level, one could use the type system (and the parser) to distinguish one from the other. The parser may factor in that some uses of function names may be syntactically forbidden for variable names and/or vice versa. Like `call f (x)` may, in some language be syntactically allowed only if `f` is the name of a function and `x` the name of a variable~. So therefore also `call f (f)` make be tractable, with two kinds of `f` declared at the same level (though it's probably ill-advised to exploit that freedom).

Such "dual-use" (or multiple-use) of names is also called *overloading*. Outside of compilers, for instance in writing technical texts, it's called *abuse of notation*. And, to awaken the reader's attention to an forthcoming such abuse, it's sometimes introduced with the words "in abuse of notation". Like "in abuse of notation, let n and m refer to nodes in control-flow graph", with the abuse warning because perhaps n and m had, in the text, so far be conventionally used for natural numbers. For the time being, the text re-uses the notation for nodes, hoping the reader can figure it out. Perhaps the writer would not just write n_1,

n_2, or n_j referring to different nodes, but even using n_n, referring to the n'th node using both interpretations of n in the same scope (i.e., chapter or section, hoping the reader can figure out which is which). This is a situation comparable to the `call f(f)` in a programming language, where the compiler may have not problems to figure out which is which (and likewise not recommended).

Often, this overloading in written technical text is avoid by using different "alphabets" or indeed different portions of alphabets or conventions (like the roman alphabet $abc\ldots$ vs. $\alpha\beta\gamma\ldots$ or $\mathfrak{abc}$ vs. $\mathbb{ABC}$ and x, y, z at the end of the latin alphabet for variables and $a, b, c\ldots$ from the beginning for constanst. In this script we also use other typographic conventions (bold-facing etc.) to help disambiguation to some extent.

In programming languages as well as in texts a certain amount of overloading occurs, sometimes even unnoticed and not every overload situation will be introduced with an abuse-warning; it's often clear enough anyway. A modest and careful use of overloading can help understanding a text or a piece of source code. It depends also on the background of the audience. Especially for beginners of some field, when too many concepts are used with the same name, it's can be confusing ("strange, earlier, if I remember correctly, XXX refered to such-and-such, but now, this XXX is used strangely, that makes no sense. What then **is** XXX **really** then, why can't the author give me a definite and unumbiguous definition?". If familiar with the concept or in very obvious situations, one would not think twice, maybe not even notice.

We will pick up on overloading when discussing types and type checking later. Different uses of the same name will generally carry diffent types, and the type system typically assists in disambuating the notation. Overloading is then discussed as one particular form of *polymorphism*, a property of type systems.

Sequential vs. simultaneous (or collateral) declarations

Now to the distinction illustrated in the following with a number of examples (in different languages). Now we do declarations not using the same name multiple times.

Many languages insist when using a name, the name must be declared, one cannot use undeclared names. Often it means, one need to declare something *before* one can use it. "Before" means roughly earlier in the program text, but only roughly so, since there is also the issue of scopes. Being declared in an earlied line in the program does not mean one can make use of the declared variable until the last line of the program, since the scope may not cover the code till then end of course. Like a local variable inside a function or a method of course can be referred to only inside the function or method body, not forever after being locally declared.

But inside one flat scope, some declarations appear earlier than others, and there is a clear notion of "before" and "after" since we assuming a *same-level* situation inside one scope.

If one declares one item after the other, then it's natural that, what is declared first can be use the subsequent declarations. That is clearly a **sequential** form of declarations and corresponds to a declare-before-use pattern.

Alternatively, declarations can be treated as occurring all at the same time. This is called a **collateral** or **simultanous** form of declarations. In a setting like that, the declaration occuring second can make use of what's been declared first in the code, and als vice versa. That allows mutually **recursive** declarations and defintions. Thus, it's an quite important form of declarations.

Also when defining, for instance, a function using *direct* recursion (not just indirect), there is the aspect of simultaneousness. Then defining resp. declaring the function, the body of the function already mentions the function, which is "currently" being introduced. That's the very nature of a recursion definition.

Listing 6.2 shows a sequention way of declarations (and definitions) in C. The variable `i` is declared and defined twice; for simplicity, let's calle it just "defined twice" (C would allow declarations also without assigning a value at the same time, hence the distinction).

The second definition is still in the same scope, but not at the same level. Therefore, it's two different variables (both called `i`) residing at different locations in memory. The first one in the global, static part of the memory, the second one on the stack. Indeed, the local variable can exists in multiple incarnations at run-time, in different activation records (or stack frames) of the procedure. We will have a closer look at that in the chapter about run-time environments. Now to the real point of the example in Listing 6.2, the sequential treatment of definitions, namely those at the local level of the function. Being *sequential*, the definition of `j` refers to the prior local definition of `i`, which means `j` obtains the value 3.

```
int i = 1;
void f(void)
  { int i = 2, j = i+1,
      ...
  }
```

Listing 6.2: Sequential declarations in C

Simultaneous or collateral definitions work differently. See for instance the example from Listing 6.3 (in ocaml). The example is comparable to the one in C (though without making use procedure-local definitions). The simultaneous declarations of (the second mentioning of) `i` and of `j` is indicated by the keyword `and`. The first definition of `i` and the simultaneous definitions of `i` and `j` are treated *sequentially*, as in the C-example.

Since the second `i` and the `j` are defined simultaneously, the `j` uses the value of `i` from before, and thus `j` is defined to contain 2.

```
let i = 1;;
let i = 2 and y = i+1;;

print_int(y);;
```

Listing 6.3: Simultaneous declarations in ocaml/ML

6.4.5 Recursive declarations/definitions

Recursive definitions and declarations are important and common. Functions, procedure, and methods are often defined recursively; via direct recursion or indirectly or mutually recursively.

Recursion can also happen at type level. For intance, the order in which classes are introduced in a program does not play a role. This means, a definition of one class, say C_1, can mention a second class C_2, when defining its members, declaring their types, and conversely, the definition of C_2 can do the same, mentioning C_1. So there is no “order” between the classes. At least not an order in the sense discussed here (sequential); there may be an inheritance order between classes, but that’s a different issue. In Java, each public class resides in its own source code file, already for that reason, there is no order between their definition in terms of their mentioning in some lines of code.

Back to functions or procedures: Listings 6.4 and 6.5 illustrate direct resp. indirection recursive definitions.

```
int gcd(int n, int m) {
  if   (m == 0) return n;
  else return gcd(m,n % m);
}
```

Listing 6.4: Direct recursion in a function defintion

```
void f(void) {
    ... g() ... }
void g(void) {
  ... f() ...}
```

Listing 6.5: Indirect recursion in a function definition

As discussed, a recursive definition is connected with the *simultaneous* treatment of declarations. For direction recursion, as in the `gcd` example, it simply means that when the body of the `gcd` function is treated, information about the `gcd`, like declaring its input and output types must already been available, i.e. entered into the symbol table. That’s not a big deal; the procedure header is mentioned (and can be treated) before the body anyway. So that feels more like a sequential situation anyway. As an aside: that it can be treated in a sequential manner is also helped by the fact that the procedure headed declares the types of the two formal parameters explicitly to be integers, as well as the return type. In a language, which allowed the programmer to leave out those explicit type declarations, things would get more involved.

But mutual recursive situations ome *really* needs to treat the definitions as collateral or simultaneous. Languages primarily working with sequential declarations (or slightly old-fashioned languages) may resort to some “tricks” resp. expect some assistance from the programmer. For instance, in the Example from Listing 6.5, the compiler my insist on a little help from the programmer to add crucial type information about `g` before `f`. So `g` is “declared” first which deos not require mentioning `~f`, then `f` is (declared and) defined, and finally, `g` is defined in a way consistent with its (and `f`’s) declaration. This is sometimes called a `prototype` (which I thinks is not a good terminology, because it can mean other things as well).

```
void g(void);  /* function prototype decl. */

void f(void) {
    ... g() ... }
void g(void) {
  ... f() ...}
```

Listing 6.6: Indirect recursion in a function definition (prototype)

In Pascal, an analogous mechanism is known as *forward declaration*. Other languages would treat all function definitions (inside a block, inside a module, or similar) as (potentially) mutually recursive. The compiler will figure it out without being alerted by special forward syntax. Still other languages use special syntax for simultaneous definitions which is used for mutually recursive function definitions. Go is a language that allows recursion without requiring speal syntax or warnings from the user, ocaml and some other functional languages use special syntax. We have seen the use of `and` earlier already.

```
func f(x int) (int) {
        return g(x) +1
}

func g(x int) (int) {
        return f(x) -1
}
```

Listing 6.7: Mutal recursion (Go)

```
let rec  f (x:int): int  =
    g(x+1)
and g(x:int) : int =
    f(x+1);;
```

Listing 6.8: Mutal recursion (ocaml)

6.4.6 Static vs. dynamic scope and binding

So far we have focused on languages with block-structured lecixal scope, simulatanous definition or otherwise. Later, in the chapter about run-time environments, we will likewise focus on there and how lexical or static binding is arranged for. That focus is justified in that the concept of lexical, nested scopes is *central*. It does not mean lexical scopes is the only way. An alternative to static (or lexical) scoping is, not surprisingly *dynanic scoping*. Connected to that is are the concepts of *static* vs. *dynamic* binding; bindings occur within a scope. When scopes are nested, a binding “belongs” to more than one scope.

We will discuss the issue of static vs. dynamic binding mostly with examples involving variables occuring in blocks of scopes. The question of static vs. dynamic binding also occurs elsewhere, for instance in class-based object-oriented languages like Java, *methods* are dynamically bound as def, and the dynamicity of the binding is wrt. to “blocks” in which the methods are mentined. We will have a look at dynamic method binding on object-oriented languages as well later.

C is a language with static scopes. Let’s have a look at Listing 6.9.

```c
#include <stdio.h>

int i = 1;
void f(void) {
  printf("%d\n",i);
}


void main(void) {
  int  i = 2;
  f();
  return 0;
}
```

Listing 6.9: Static scoping in C

The code contains two definitions of `i`, a global one and one local to `main`. The declaration of `f`, i.e., the use of `i` in `f`'s body refers to the global instance of `i`, since `f` defined on the global level. Additionally, the definition of `i` is done *sequentially* before `f` and the body of `f` does not contain a local (re-)definition of `i` that would overshadow the global one. That's the situation concerning the *definition* of `f`.

Now to the *use* of `f`. It's called only one time, inside `main` and in a scope that works with a local definition of `i`. When executing `f`, to which declaration does the printed `i` binds to, which is the relevant scope for it?

Using static binding means, relevant is the scope where "statically" the function was defined, its static scope (in this example the global one). What is printed therefore is 1, of course.

Dynamic binding (for variable `i` again) is illustrated in Figure 6.10. In this example, the assignment inside `Q` affects the variable `i` as introduced in line 4, since this is the relevant scope in which `Q` is *called*. The procedure `Q` is called only once in the example. If there were different calles originating from inside different scopes, different `i`'s may be affected. For static scopes, which `i` is meant is statically fixed from the place where the function is defined.

```
 1  void Y () {
 2    int i;
 3    void P() {
 4      int i;
 5      ...;
 6      Q();
 7    }
 8    void Q(){
 9      ...;
10      i := 5;  // which i is meant?
11    }
12    ...;
13
14    P();
15    ...;
16  }
```

Listing 6.10: Dynamic scoping (pseudo code)

Let's look at some non-pseudo code examples. Let's take TeXor LaTeX. Those are more domain-specific languages rather than general purpose languages; though TeXallows loops, conditionals etc, so those languages are Turing complete. Also it's clearly a compiler, translating some textual source language into some output format, like dvi, ps, or pdf.

TeXand LaTeXalso make use of scopes; one does not have to work with one global scope. In the examples from Listing 6.11 and 6.12, the beginning and the end of an inner scope is marked by { and } (there are other ways to obtain scopes in LaTeX, but it's unimportant for us). Both examples basically the same (using the slightly different syntax of TeXand LaTeX). What happens is very easy to check by invoking TeX or LaTeX and check the genrated output, say pdf.

```
\def\astring{a1}
\def\x{\astring}
\x
{
  \def\astring{a2}
  \x
}
\x
\bye
```

Listing 6.11: Dynamic scoping (TeX)

```
\documentclass{article}
\newcommand{\astring}{a1}
\newcommand{\x}{\astring}
\begin{document}
\x
{
  \renewcommand{\astring}{a2}
  \x
}
\x
\end{document}
```

Listing 6.12: Dynamic scoping (LaTeX)

The next illustration uses Lisp. In particular, emeacs lisp. If one has access to emacs, also that is easy to run and check, simply firing off emacs and evaluate the lisp snippet in a buffer, for instance.

The are many lisp dialects, emacs lisp just one of theme. Another important one is Scheme, actually Scheme was the first one (or at least the first significant and most prominent one) with *static* or lexical scoping. Originally, Lisp used dynamic binding. Lisp was way ahead of its time in some ways, actually revolutionary (higher-order functions, reflection, garbage collecton), one should not forget that it was conceived (and implemented!) in the 50ies (at MIT). Now, resource requirements for Lisp stretched the hardware and compiler concepts of those days. Note that the very earliest machines did not even have hardware support for stack pointers (Borroughs machines at the beginning of the 60ies where the first that pioneered that) which made even recursion (which uses stack) a costly luxury. And Lisp supported higher-order functions from the start. It took some time (and conceptual and hardware advances) until major lexically-scoped variant of Lisp could establish itself (known as Scheme). Scheme also supports dynamic scoping (though frowns upon it). More "classic" Lisp dialects (like Common Lisp) also support lexical scoping besides dynamically scoping in the meantime.

```
(setq astring "a1")  ;; ``assignment''
(defun x() astring)  ;; define ``variable x''
(x)                  ;; read value
(let ((astring "a2"))
      (x))
```

Listing 6.13: Dynamic binding in elisp

Emacs Lisp is one well-known Lisp-dialect based on dynamic scoping, though as of emacs version 24, also lexical scoping is supported. It may or may not be a coincidence, that the key person behind *emacs* is Richard Stallman, the last of the “last true hacker” from the MIT school of hackers. McCarthy and Minsky were also at the MIT (earlier pioneers than Stallman), McCarthy is central behind Lisp, and actually also coined the term AI (MIT was a if not the focal point of early AI). For emacs, Stallman is central in kicking it off, hacking its initial versions (with others), mentoring it through many years and giving a spiritual (or idealogical?) background as part of a larger free software movement.

To round off the discussion, let’s go back to static binding and point somthing out that probably should be clear anyway, but it can’t hurt to rub it in. As said, in static binding, it’s the static scope “that counts”, where a variables has been declared. But it’s *not* about the original value. A value may change, and as far as the *value* is concerned, static binding does *not* mean, when used inside a function, for instance as in a situation as in Listing 6.9, that it’s the original value that counts. Static binding refers to the association of a variable with a memory location or address, not the association with a particular value.

Listing 6.14 illustrates that, in this case using Go, not C. Both languages use static binding of variable. As can be seen in the example, unlike C, Go supports nested function definitions.

```
package main
import ("fmt")

var f = func ()   {
  var x = 0
  var g = func() {fmt.Printf(" x = %v", x)}
  x = x + 1
    {
      var x = 40                  // local variable
      g()
      fmt.Printf(" x = %v", x)}
}
func main() {
  f()
}
```

Listing 6.14: Static binding and mutating values (in Go)

The value of `x` printed in the body of `g` is 1, not `0` which is the value at the point when `g` is defined. Overally, what is printed is `x = 1 x = 40`.

```
package main
import ("fmt")

var f = func () (func (int) int)  {
        var x = 40                  // local variable
        var g = func (y int) int { // nested function
```

```
                return x + 1
        }
        x = x+1                      // update x
        return g                     // function as return value
}

func main() {
        var x = 0
        var h = f()
        fmt.Println(x)
        var r = h (4)
        fmt.Printf(" r = %v", r)
}
```

Listing 6.15: Static binding and higher-order functions (in Go)

Maintaining lexical binding can become challenging. The Example from Listing 6.15 basiscally does the same as the previous one: static binding and changing the value (the latter part, however, changing the value is not central to the purpose of the example).

It is more complex than the previous one in that it not just use nested function definitions, but makes use of *higher-order* functions. That's another feature supported by Go, but not by C. It does not change what static-binding means, it just makes it harder for the compiler to achieve. We will discuss that in slightly more detail in the chapter about run-time environments. Here we just point out what is responsible for the complications

As said, the example uses higher-order functions. In particular, the function `f` gives back some function, namely the function `g`, and not only that: function `g` is defined *inside* `f`, in particular, `g` is defined inside the scope of `f`. And finally, the nested function `g` refers to `x`, which is also defined inside `f`. Now the problem is that the scope of `f` lives longer than the body of `f` itself. In many languages, one important part of the RTE is the run-time stack, or call stack. It turns out, that in situations like the ones illustrated here, a stack is no longer good enough for providing lexical scoping `f`.

At any rate, lexical scoping in the example results that `r = 42` is printed.

6.5 Symbol tables as attributes in an AG

Let's have a short look at how to represent symbol tables (in an easy setting) with attribute grammars. We illustrate it on an fragment of a language covering expressions and declarations.

In this short section, we want to illustrate symbol tables by attributed grammars. Not how they are implemented, but how they are connected to traversing the abstract syntax tree and that connection is specified by semantics rules.

We have seen similar syntax before, for intance also in the chapter about attribute grammars. There we have seen attribute grammar examples for evaluating expressions and another example, dealing with declarations. The first one used synthesized attributes, the second one mostly inherited ones. The example here does not involve expression evaluation, it focuses in the declaration part, i.e., checking if an variable has been declared earlier. In a typical setting that would also involve type checking, i.e., not just checking when using a variable whether it has been declared before, but that the use of the variable is consistent type-wise with its earlier declaration. Also that typing aspect is absent in the example, but it would be straightforward to add.

With or without type-checking, to check conformity in this setting, the information flows is from the declarations to the uses of a variable. Since the declarations (here) come before the uses, it means, the declarations are higher-up in the syntax tree and the information therefore “flows” downwards. In other words, we are dealing conceptually with *inherited* attributes. We have seen that in the chapter about attribute grammars before. The grammar will not be solely on inherited attributes, there will also be synthesized ones.

Still, symbol tables can be seen as a realization of *inherited attributes*. At least in a simple situation like the one here, with a “declare-before-use” regiment. As discussed, there are more complex forms of declarations, notably those allowing recursion. Of course, also with recursive declarations, one can use symbol tables. Likewise one can also capture those more complex situations by attribute grammars. As discussed, for attribute grammars, cycles in the so-called dependency graphs of parse trees are strictly forbidden. Still one could capture recursive declarations by formalizing a “staged approach” and with additional attributes, like first adding partial information and then going through the abstract syntax tree a second time. In an attribute grammar, that may be done splitting a symbol-table attribute into an attribute capturing the preliminary stage with partially entered information and then the final version. Using two different attributes for that would break the forbidden cyclic dependencies. This way of approaching declaration-checking (or type checking) correspond also the way how one would do it working directly with symbol tables in an implementation (without considering it as an attribute grammar problem).

We don’t cover recursive declarations, our attribute grammar will therefore be simpler. The small digression about recursion is added to dispell a possible (wrong) impression, that attribute grammars can only capture declare-before-use situations, due to their acyclicity restriction.

The example, however, deals with one important complication, namely *nested scopes*. This aspect was *not* covered yet in the earlier chapter about attribute grammars. The syntax we will be dealing with can similarly found in various languages. A small piece of code in ocaml is shown in Listing 6.16; remember also the code from Listing 6.3, used earlier when discussing simultanous vs. sequential declarations.

```
let x = 2 and y = 3 in
  (let x = x+2 and y =
        (let z = 4 in x+y+z)
   in print_int (x+y))
```

Listing 6.16: Nested lets (in ocaml)

6.5.1 Expressions and declarations: grammar

Let’s start fixing the syntax by giving the grammar in BNF. We

$$\begin{array}{rcl}
S & \rightarrow & \mathit{exp} \\
\mathit{exp} & \rightarrow & (\,\mathit{exp}\,) \mid \mathit{exp} + \mathit{exp} \mid \mathbf{id} \mid \mathit{num} \mid \mathbf{let}\ \mathit{dec\text{-}list}\ \mathbf{in}\ \mathit{exp} \\
\mathit{dec\text{-}list} & \rightarrow & \mathit{dec\text{-}list}\mathbf{,}\ \mathit{decl} \mid \mathit{decl} \\
\mathit{decl} & \rightarrow & \mathbf{id} = \mathit{exp}
\end{array}$$

We want the following informal *rules* what's allowed and what's not for declarations. Those need to be afterwards captured by semantic rules of the attribute grammar:

1. No identical names in the same let-block,
2. used names must be declared,
3. most-closely nested binding counts, and
4. *sequential* (non-simultaneous) declaration (≠ ocaml/ML/Haskell . . .)

These rules are illustrated in Listing 6.17. Note that we intend to use a *sequential*, not a simultaneous interpretation of declarations. In the exercises, one task will be to port the sequential treatment here to *simultaneous* declarations.

```
let x = 2, x = 3 in x + 1          (* no, duplicate  *)

let x = 2  in x+y                  (* no, y unbound *)

let x = 2 in (let x = 3 in x)      (* decl. with 3 counts *)

let x = 2, y = x+1                 (* one after the other *)
in  (let x = x+y,
         y = x+y
     in y)
```

Listing 6.17: Illustration of what's allowed and what not

The attributes used in the grammar are shown in Table 6.1. We also indicate which ones are inherited and which are synthesized. Note the special "status" of the attribute of the terminal symbol **id**. It has a special status in that we assume it injected by the scanner. The issue of attributes of terminals, whether should or should not be inherited, synthesized, or something else has been discussed in the attribute grammar chapter.

Let's discuss two further aspects of the attributes. Both aspects have to do with the fact that attribute grammars are a functional, declarative formalism, working with equations on attributes.

One aspect is *errors* and *error handling*. For the symbol table task, there will be situations that correspond to errors. In an implementation that may covered by raising an *exception* (and perhaps handling it). Exceptions and their treatment is not part of attribute grammars, in particular the fact that raising an exception means, breaking out of the normal, i.e., *un-exceptional* control flow. Exceptions can be part of the interface of the symbol-table. For instance, the attempt to look up a variable which has not been entered may result in a specific exception, likewise trying to enter a double binding at the same nesting level. That way, the programmer of the definedness-checker (or type checker) would not have to write code to check whether the mentioned conditions are met; the symbol-tables makes sure of that and maintain a corresponding invariant by themselves. Perhaps that's a more robust design, but of course the programmer of the checker still needs to write code to *handle* the exceptions properly, for instance like translating the symbol-table exception into more helpful user error message (which again could be done by catching the symbol-table expection and letting the handler (re-)raise another, more informative exception).

In general, evaluating a dependency graph for a parse tree corresponds to a tree traversal, in our setting, with mostly inherited attributes, a traversal “downwards” (and upwards afterwards, using a recursive procedure, like depth-first traversal). But there is no mechanism that would allow to stop the tree traversal after stumbling over an error (like an undeclared variable or a multiple declaration of the same variable at the same level). Instead the traversal has to continue, and one needs to somehow “simulate” the exceptional situation using attributes. That explains the error-attribute. The attribute is synthesized, since the error condition propagates from the place where it occurs up to the root. That’s all fine and not actually complicated. However, to handle the non-erronous and the erroneous situation, the “user” of the attribute grammar has to add boiler-plate code all over the place (“if not errow do this else do that”). That clutters the semantic rules and makes the solution slightly unelegant (see the attribute grammar below). We will make a similar remark later in the chapter about type checking (where there is also a section that specifies a simple type system using an attribute grammar).

The second aspect I want to discuss is the treatment of the symbol table. We said that symbol tables somehow corresponds to a (mostly) inherited attribute. That’s not incorrect. However, Table 6.1 shows that the attribute grammar has *3 attributes* to capture the symbol table. Partly that’s caused by the fact that the attributes are on different (non-terminal) symbols. In particular `symtab` is an attribute of *exp*, whereas `intab` and `outtab` are attributes of *dec-list* resp. *decl*. That makes `symbtab` a different attribute from the other two in some way, and we might choose to simply rename `symbtab` to `intab` or `outtab` without changing the solution (though probably `symbtab` is a clearer choice). Anyway, it’s not the point I want to make, the point is about the attributes of *decl* and *dec-list*, `intab` and `outtab`. There, we really need to have two *different* attribute names. Declarations and declaration lists *change* the symbol table insofar that new binding(s) are added (unless an “exception” occurs).

That means, the state of the symbol table before the declaration or before a list of declaration is typicalled different from the state afterwards. This is captured by the two different attributes, `intabl` and `outtab`. In many languages, the symbol table would conventionally implemented imperatively (though also efficient functional implementations like using red-black trees) exist. For instance, the hash-tables which often underly symbol-tables are conventionally an imperative data structure. That means, in an implementation, handling a declaration is an operation that *changes* the symbol-table. In the attribute grammar here, we specify how a declaration transforms the symbol-table from the state before (`intab`) to its state afterwards (`outtab`, because we are working with equations, which are side-effect free.

Attributes in attribute grammars are generally typed. We don’t explicitly list the types in a separate table; they should mostly be clear: nesting level is an integer, actually a non-negative one, the outermost nesting is counted as level 0, as the attribute grammar shows. The error attribute is a kind of boolean: is there an error, yes or no? As said, in a practical situation (with or without exception), one might choose to refine it with information about what kind of error occured and/or where. The most complex data

symbol	attributes	kind
exp	`symtab`	inherited
	`nestlevel`	inherited
	err	synthesized
dec-list, *decl*	`intab`	inherited
	`outtab`	synthesized
	`nestlevel`	inherited
id	`name`	injected by scanner

Table 6.1: Attributes for the symbol tables

structure, of course, is the symbol table itself. For that Table 6.2 contains the *interface* and that's is type-related information.

We see in particular in the signature of the `insert` function that it returns a (changed) symbol table, instead of changing the state of the argument `tab` in-place.

return type		
symboltable	`insert(tab,name,lev)`	returns a changed table
bool	`isin(tab,name)`	boolean check
int	`lookup(tab,name)`	gives back *level*
symboltable	`emptytable`	you have to start somewhere
	`errtab`	erroronous table

Table 6.2: Interface of the symbol table

Treatment of nested scopes here

A few words also on the "design" of the symbol tables in connection with nested scopes. In earlier sections, we discussed the issue to some extent (chained symbol tables or specific arrangements in hash-table).

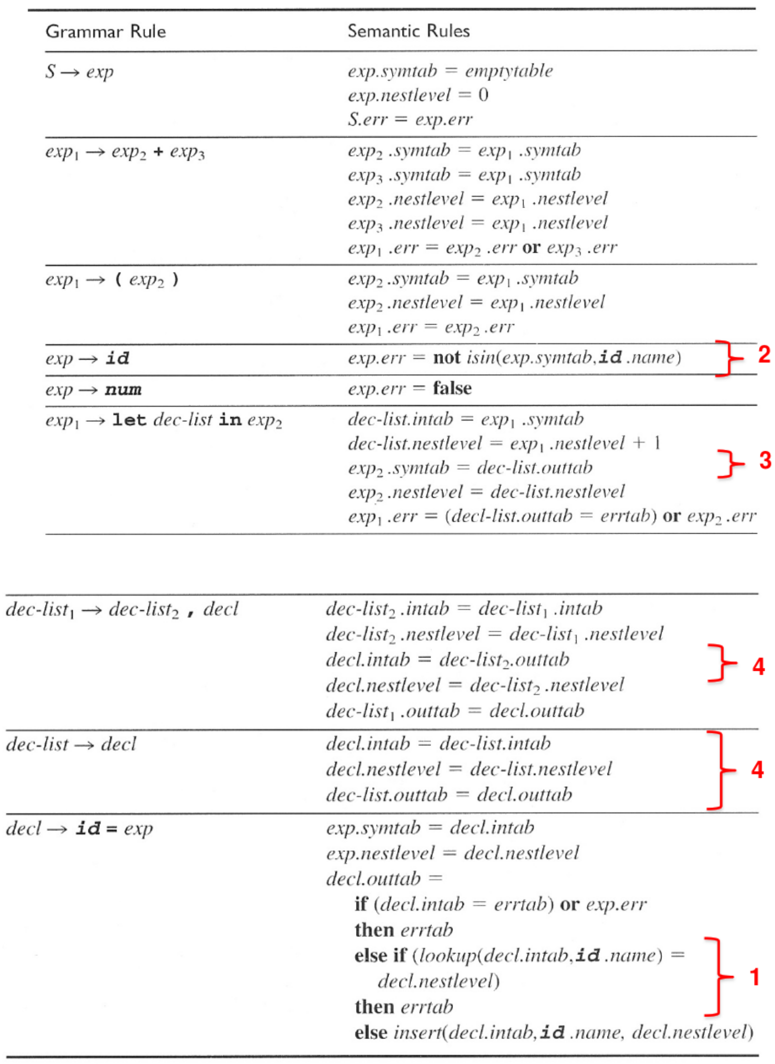
Here, the attribute grammar operates with nesting levels. The nesting level is explicitly handed over as argument when entering a binding in `insert`. That's another way of dealing with nested block structures.

One central production in that context is, of course, the one dealing with let-declarations. The production and the semantic rules dealing with the nesting level are shown in equation (6.1).

$$exp_1 ::= \textbf{let}\ dec\text{-}list\ \textbf{in}\ \ exp_2 \qquad \begin{array}{l} dec\text{-}list.\texttt{nextlevel} = exp_1\,.\texttt{nextlevel} + 1 \\ exp_2\,.\texttt{nextlevel} = dec\text{-}list.\texttt{nextlevel} \end{array} \tag{6.1}$$

When processing the let-declaration, the nesting level is increased by one for *both* the declaration list and the body exp_2 of the declaration. Assume that exp_1 occurs at a nesting depth of n. That exp_2 is processed at a nesting level $n + 1$ is clear.

For *expressions* in a *dec-list*, they conceptually are occuring at level n, at the *same* level than exp_1, the next nesting level kicks in only in the body. So the level $n+1$ for *dec-list* is to be interpreted as "use $n+1$ as level when adding a new declaration", thus building up (at level n) the bindings *for* the body at level $n+1$.



Grammar Rule	Semantic Rules	
$S \rightarrow exp$	$exp.symtab = emptytable$ $exp.nestlevel = 0$ $S.err = exp.err$	
$exp_1 \rightarrow exp_2$ **+** exp_3	$exp_2.symtab = exp_1.symtab$ $exp_3.symtab = exp_1.symtab$ $exp_2.nestlevel = exp_1.nestlevel$ $exp_3.nestlevel = exp_1.nestlevel$ $exp_1.err = exp_2.err$ **or** $exp_3.err$	
$exp_1 \rightarrow$ **(** exp_2 **)**	$exp_2.symtab = exp_1.symtab$ $exp_2.nestlevel = exp_1.nestlevel$ $exp_1.err = exp_2.err$	
$exp \rightarrow$ **id**	$exp.err =$ **not** $isin(exp.symtab,$**id**$.name)$	2
$exp \rightarrow$ **num**	$exp.err =$ **false**	
$exp_1 \rightarrow$ **let** *dec-list* **in** exp_2	$dec\text{-}list.intab = exp_1.symtab$ $dec\text{-}list.nestlevel = exp_1.nestlevel + 1$ $exp_2.symtab = dec\text{-}list.outtab$ $exp_2.nestlevel = dec\text{-}list.nestlevel$ $exp_1.err = (decl\text{-}list.outtab = errtab)$ **or** $exp_2.err$	3
$dec\text{-}list_1 \rightarrow dec\text{-}list_2$ **,** $decl$	$dec\text{-}list_2.intab = dec\text{-}list_1.intab$ $dec\text{-}list_2.nestlevel = dec\text{-}list_1.nestlevel$ $decl.intab = dec\text{-}list_2.outtab$ $decl.nestlevel = dec\text{-}list_2.nestlevel$ $dec\text{-}list_1.outtab = decl.outtab$	4
$dec\text{-}list \rightarrow decl$	$decl.intab = dec\text{-}list.intab$ $decl.nestlevel = dec\text{-}list.nestlevel$ $dec\text{-}list.outtab = decl.outtab$	4
$decl \rightarrow$ **id** **=** exp	$exp.symtab = decl.intab$ $exp.nestlevel = decl.nestlevel$ $decl.outtab =$ **if** $(decl.intab = errtab)$ **or** $exp.err$ **then** $errtab$ **else if** $(lookup(decl.intab,$**id**$.name) =$ $decl.nestlevel)$ **then** $errtab$ **else** $insert(decl.intab,$**id**$.name, decl.nestlevel)$	1

Figure 6.1: Attribute grammar

Bibliography

Index